Experiment-2

Feedforward Neural Network (MLP)

1. Aim

To understand the architecture and training process of a Multilayer Perceptron (MLP) for tabular data by implementing it on the Iris dataset, with detailed visualization of forward propagation, backpropagation, and hidden-layer activations for selected samples.

2. Theory

Deep feedforward networks — often called feedforward neural networks or multilayer perceptrons (MLPs) — are fundamental models in deep learning. The goal of a feedforward network is to approximate some target function f *. For example, for a classifier, y = f *(x) maps an input x to a category y.

A layered feedforward network is one in which any path from an input node to an output node traverses the same number of layers. For example, the n^{th} layer of such a network consists of all nodes that are n edge traversals from an input node. A hidden layer is any layer that is neither the input nor the output layer. A network is fully connected if each node in layer i is connected to all nodes in layer i+1. Layered feedforward networks have become popular because they often generalise well: when trained on a relatively sparse set of examples they frequently provide correct outputs on unseen test data.

When we use a feedforward neural network to accept an input x and produce an output yˆ, information flows forward through the network. The input x provide the initial information that then propagates to the hidden units at each layer and finally produces yˆ. This is called forward propagation.

During training, forward propagation continues until it produces a scalar cost J(θ). The backpropagation algorithm (Rumelhart et al., 1986), often simply called backprop, allows information from the cost to flow backwards through the network to compute gradients. The backpropagation algorithm that trains the feedforward neural network can often find a good set of weights (and biases) in a reasonable amount of time. Backpropagation relies on the chain rule to compute derivatives and typically uses a least-squares or cross-entropy criterion depending on the task.

Layer-by-Layer (MLP) Transformation:

- **Input Layer:** Receives raw input x.

- **First Hidden Layer: $h_1 = g_2$ ($W_1 \cdot$ $x + b_1$)** Applies a linear transformation followed by a nonlinearity.
- **Second Hidden Layer (if any): $h_2 = g_2(W_2 \cdot$ $h_1 + b_2)$** Further transforms the representation
- **Output Layer: $y = g_n(W_n \cdot$ $h_{n-1} + b_n)$** Produces the final predictions.

More generally, the network can be expressed as a composition of functions:

$\Phi(x) =$ **$W_n \rho \cdot$ $W_{n-1} \cdot$ .... $\cdot$ $\rho \cdot$ $W_1(x)$**

Where:

- **$W_i = A_i \cdot x + b_i$** represents the linear transformation (weights and biases) for layer i.
- **ρ** is the activation function (often the same across layers).

Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called feedforward neural networks. This means there are no loops in the network, thus information is always feed forward, never feedback which can also be shown as in Figure 1.

These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself.

I. **Gradient Based Learning:** For feedforward neural networks, it is important to initialise all weights to small random values; biases may be initialised to zero or to small positive values. Iterative gradient-based optimisation algorithms (e.g., SGD, RMSprop, Adam) are used to train feedforward networks and deepest models.

II. **Learning XOR:** To illustrate the capabilities of feedforward networks, consider the XOR function. XOR returns 1 when exactly one of x_1 or x_2 is 1, and 0 otherwise. Learning XOR demonstrates that an MLP with a hidden layer can represent non-linearly separable functions.

To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function. The XOR function ("exclusive or") is an operation on two binary values, x_1 and x_2. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0. The XOR function provides the target function $y = f^*(x)$ that we want to learn. Our model provides a function $y = f(x;\theta)$ and our learning algorithm will adapt the parameters θ to make f as similar as possible to f*.

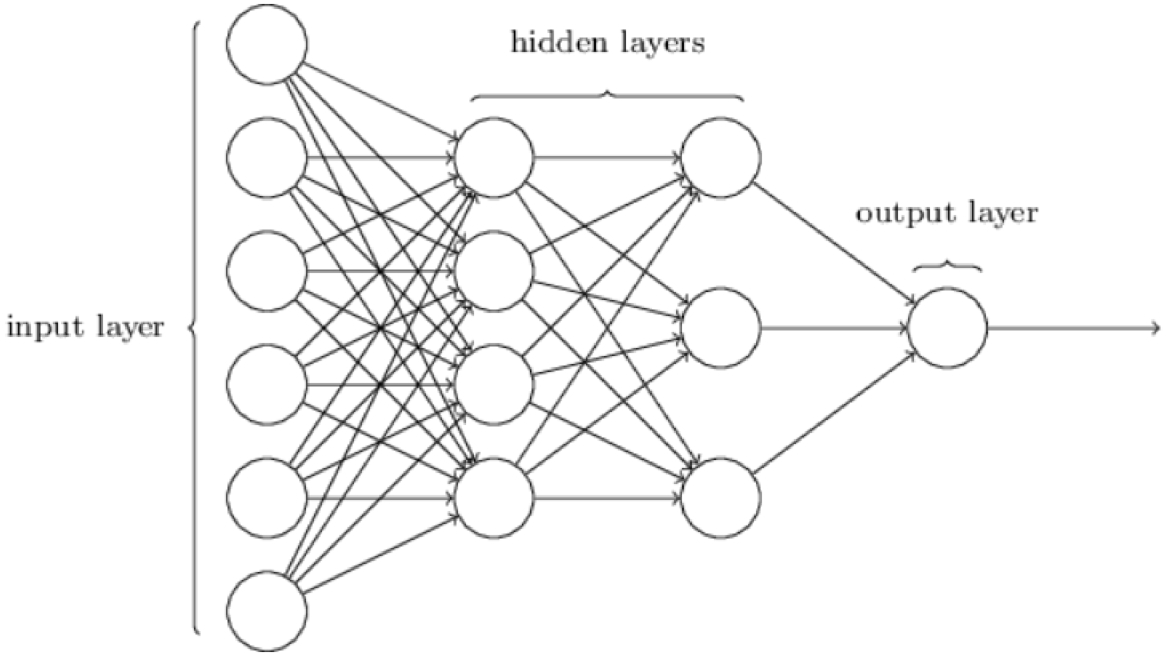


Figure 1- Architecture of Feedforward Neural Network

(Source: M. A. Nielsen, Neural Networks, and Deep Learning.)

The process of forward propagation from input to output and backward propagation of errors is repeated several times until the error gets below a predefined threshold. The whole process is represented in the following diagram:

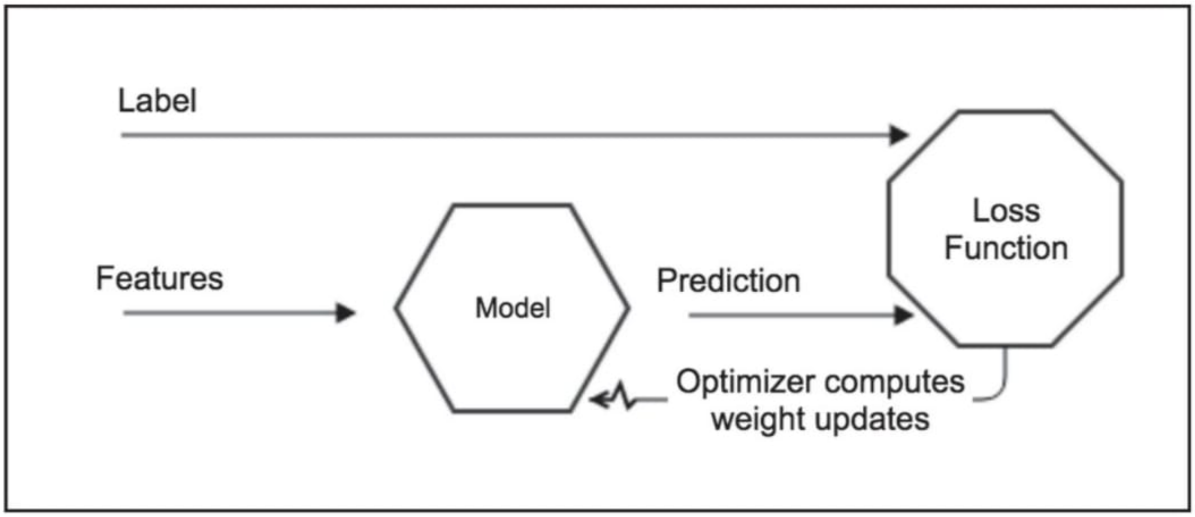


Figure 2- MLP Process both Forward and Backpropagation

(Source: Antonio Gulli, Sujit Pal, Deep Learning with Keras)

The forward and backward propagation process is repeated until the error falls below a predefined threshold. The model is updated to progressively minimise the loss function. In a neural network, individual neuron outputs matter less than the collective behaviour of weights in each layer as shown in Figure 2; the network adjusts its internal weights, so the prediction accuracy increases. Using appropriate features and high-quality labels is fundamental for reducing bias and improving learning.

Merits of Feedforward Neural Network (MLP):

- **Scalability:** The number of hidden layers and neurons can be adjusted to match problem complexity.
- **Performance on tabular data:** For many structured datasets, MLPs can outperform more complex models due to their simplicity and ability to learn direct features.
- **Universal function approximation:** MLPs can approximate virtually any continuous function, enabling them to model complex, non-linear relationships.

Demerits of Feedforward Neural Network (MLP):

- **Sensitivity to hyperparameters:** Performance depends heavily on choices such as number of layers, units, learning rate, and activation functions.
- **Overfitting:** MLPs can memorise training data and generalise poorly, especially with small datasets.
- **Gradient issues:** Very deep networks may encounter vanishing or exploding gradients, making training difficult.
- **Data requirements:** Large amounts of labelled data are often necessary to train effectively.

Code & Result:

Step-1: Importing Important Libraries

```
[ ]: # ---------------- Part 1: Importing Libraries ----------------
     import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import seaborn as sns
     sns.set()

     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler, LabelBinarizer
     from sklearn.metrics import classification_report, confusion_matrix,␣
      ↪accuracy_score

     import tensorflow as tf
     from tensorflow.keras import Model
     from tensorflow.keras.layers import Dense, Input

     # small helper
     import json, os, time
     print("Libraries imported. TensorFlow version:", tf.__version__)
```

```
Libraries imported. TensorFlow version: 2.19.0
```

Step-2: Dataset Creation

```
[ ]: # ---------------- Part 2: Dataset Creation ----------------
     # Change FILE_PATH to your environment (Kaggle/Colab)
     FILE_PATH = "/content/Iris (2).csv"     # <--- edit if needed

     # 1) Load
     data = pd.read_csv(FILE_PATH)
     # 2) Drop Id if present
     if "Id" in data.columns:
         data = data.drop("Id", axis=1)

     # 3) Inspect
     print("Shape:", data.shape)
     display(data.head())
```

```
print("\nClass distribution:")
print(data["Species"].value_counts())

# 4) Features & labels
X = data.drop("Species", axis=1).values              # shape (N,4)
y_species = data["Species"].values                   # string labels

# 5) Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 6) One-hot encode labels
binarizer = LabelBinarizer()
y_encoded = binarizer.fit_transform(y_species)       # shape (N,3)
print("Classes (order):", binarizer.classes_)

# 7) Train/test split (stratified)
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_encoded, test_size=0.2, random_state=42, stratify=y_encoded
)
print("Train shape:", X_train.shape, "Test shape:", X_test.shape)
```

```
Shape: (150, 5)

   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm      Species
0            5.1           3.5            1.4           0.2  Iris-setosa
1            4.9           3.0            1.4           0.2  Iris-setosa
2            4.7           3.2            1.3           0.2  Iris-setosa
3            4.6           3.1            1.5           0.2  Iris-setosa
4            5.0           3.6            1.4           0.2  Iris-setosa


Class distribution:
Species
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: count, dtype: int64
Classes (order): ['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']
Train shape: (120, 4) Test shape: (30, 4)
```

```
[ ]: # ---------------- Part 3: Initializing Parameters & Model Building␣
       ↪-----------------

     # Hyperparameters (as you requested)
     EPOCHS = 100
     BATCH_SIZE = 8
     LEARNING_RATE = 0.01
     OPTIMIZER = "RMSprop"    # descriptive only
```

```python
# Model architecture (matches your PPT)
input_dim = X_train.shape[1]    # 4
num_classes = y_train.shape[1] # 3

inputs = Input(shape=(input_dim,), name="Input_Layer")
x1 = Dense(10, activation="relu", name="Hidden_Layer_1")(inputs)
x2 = Dense(8, activation="relu", name="Hidden_Layer_2")(x1)
outputs = Dense(num_classes, activation="softmax", name="Output_Layer")(x2)

model = Model(inputs=inputs, outputs=outputs, name="Iris_MLP_BP_Inspect")
# We will use a custom loop (GradientTape), so no compile() required for training,
# but create optimizer & loss func:
optimizer = tf.keras.optimizers.RMSprop(learning_rate=LEARNING_RATE)
loss_fn   =  tf.keras.losses.CategoricalCrossentropy(from_logits=False)

# Metrics (tf.metrics works across TF versions)
train_loss_metric  =  tf.keras.metrics.Mean(name="train_loss")
train_acc_metric    = tf.keras.metrics.CategoricalAccuracy(name="train_acc")
val_loss_metric     = tf.keras.metrics.Mean(name="val_loss")
val_acc_metric      =  tf.keras.metrics.CategoricalAccuracy(name="val_acc")

print("Model built with RMSprop lr=", LEARNING_RATE)
model.summary()
```

```
Model built with RMSprop lr= 0.01
Model: "Iris_MLP_BP_Inspect"


 Layer (type)                  Output Shape               Param #

 Input_Layer (InputLayer)      (None, 4)                        0

 Hidden_Layer_1 (Dense)        (None, 10)                      50

 Hidden_Layer_2 (Dense)        (None, 8)                       88

 Output_Layer (Dense)          (None, 3)                       27


 Total params: 165 (660.00 B)

 Trainable params: 165 (660.00 B)
```

```
Non-trainable params: 0 (0.00 B)
```

[ ]:

```python
# ---------------- Part 4: Training (custom loop) ----------------
# Robust metric reset helper (works across TF versions)
def reset_metric(m):
    if hasattr(m, "reset_states") and callable(getattr(m, "reset_states")):
        m.reset_states()
    elif hasattr(m, "reset_state") and callable(getattr(m, "reset_state")):
        m.reset_state()
    else:
        # best-effort zero internal variables
        try:
            for v in getattr(m, 'variables', []):
                v.assign(tf.zeros_like(v))
        except Exception:
            pass

# Prepare tf.data pipelines
train_ds = tf.data.Dataset.from_tensor_slices((X_train.astype(np.float32),
  y_train.astype(np.float32)))
train_ds = train_ds.shuffle(buffer_size=1024, seed=42).batch(BATCH_SIZE).
  prefetch(tf.data.AUTOTUNE)
val_ds = tf.data.Dataset.from_tensor_slices((X_test.astype(np.float32), y_test.
  astype(np.float32)))
val_ds = val_ds.batch(256).prefetch(tf.data.AUTOTUNE)

# Choose a test index to inspect backprop flow at checkpoints
inspect_idx = 5
x_inspect = X_test[inspect_idx].astype(np.float32)
y_inspect = y_test[inspect_idx].astype(np.float32)

# We'll store histories and gradient norms
history = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': []}
gradnorm_history = {} # keys: epoch -> {layer: L2norm}

# Get references to kernel (weight) tensors for the Dense layers (we compute
  grads w.r.t kernels)
layer_kernels = {
    'Hidden_Layer_1': model.get_layer('Hidden_Layer_1').kernel,
    'Hidden_Layer_2': model.get_layer('Hidden_Layer_2').kernel,
    'Output_Layer': model.get_layer('Output_Layer').kernel
}

start_time = time.time()
for epoch in range(1, EPOCHS + 1):
    # Reset metrics
    reset_metric(train_loss_metric); reset_metric(train_acc_metric)
    reset_metric(val_loss_metric); reset_metric(val_acc_metric)

    # Training
    for xb, yb in train_ds:
        with tf.GradientTape() as tape:
            logits = model(xb, training=True)
            loss_value = loss_fn(yb, logits)
        grads = tape.gradient(loss_value, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
        train_loss_metric.update_state(loss_value)
        train_acc_metric.update_state(yb, logits)

    # Validation
    for xb_val, yb_val in val_ds:
        val_logits = model(xb_val, training=False)
        vloss = loss_fn(yb_val, val_logits)
        val_loss_metric.update_state(vloss)
        val_acc_metric.update_state(yb_val, val_logits)

    # Record per-epoch metrics
    try:
        train_loss = float(train_loss_metric.result().numpy())
        train_acc  = float(train_acc_metric.result().numpy())
        val_loss   = float(val_loss_metric.result().numpy())
        val_acc    = float(val_acc_metric.result().numpy())
    except Exception:
        train_loss = float(train_loss_metric.result())
        train_acc  = float(train_acc_metric.result())
        val_loss   = float(val_loss_metric.result())
        val_acc    = float(val_acc_metric.result())

    history['train_loss'].append(train_loss)
    history['train_acc'].append(train_acc)
    history['val_loss'].append(val_loss)
    history['val_acc'].append(val_acc)

    # Save a checkpoint at each 10th epoch (TensorFlow requires .weights.h5 for␣
↪save_weights)
    if epoch % 10 == 0:
        ckpt_name = f'cp_epoch_{epoch}.weights.h5'
        model.save_weights(ckpt_name)

        # Compute gradient norms for the chosen single sample (separate tape)
        x_single = tf.expand_dims(tf.constant(x_inspect, dtype=tf.float32),␣
↪axis=0)
```

```
        y_single = tf.expand_dims(tf.constant(y_inspect, dtype=tf.float32), axis=0)
        with tf.GradientTape() as tape2:
            logits_single = model(x_single, training=False)
            loss_single = loss_fn(y_single, logits_single)
        grads_single = tape2.gradient(loss_single, [layer_kernels['Hidden_Layer_1'], layer_kernels['Hidden_Layer_2'], layer_kernels['Output_Layer']])
        norms = [float(tf.norm(g).numpy()) if g is not None else 0.0 for g in grads_single]
        gradnorm_history[epoch] = {
            'Hidden_Layer_1': norms[0],
            'Hidden_Layer_2': norms[1],
            'Output_Layer':   norms[2]
        }
        print(f"Checkpoint saved: {ckpt_name} | epoch {epoch} | train_loss={train_loss:.4f} val_acc={val_acc:.4f}")

    else:
        # Light progress logging
        if epoch % 10 == 1 or epoch == EPOCHS:
            print(f"Epoch {epoch:03d} | train_loss={train_loss:.4f} train_acc={train_acc:.4f} | val_loss={val_loss:.4f} val_acc={val_acc:.4f}")

# Save histories & gradnorms
with open('history_backprop.json','w') as f:
    json.dump(history, f)
with open('gradnorm_history.json','w') as f:
    json.dump(gradnorm_history, f)

print("Training complete in {:.1f}s".format(time.time() - start_time))
print("Saved checkpoints:", sorted([f for f in os.listdir('.') if f.startswith('cp_epoch_') and f.endswith('.weights.h5')]))
```

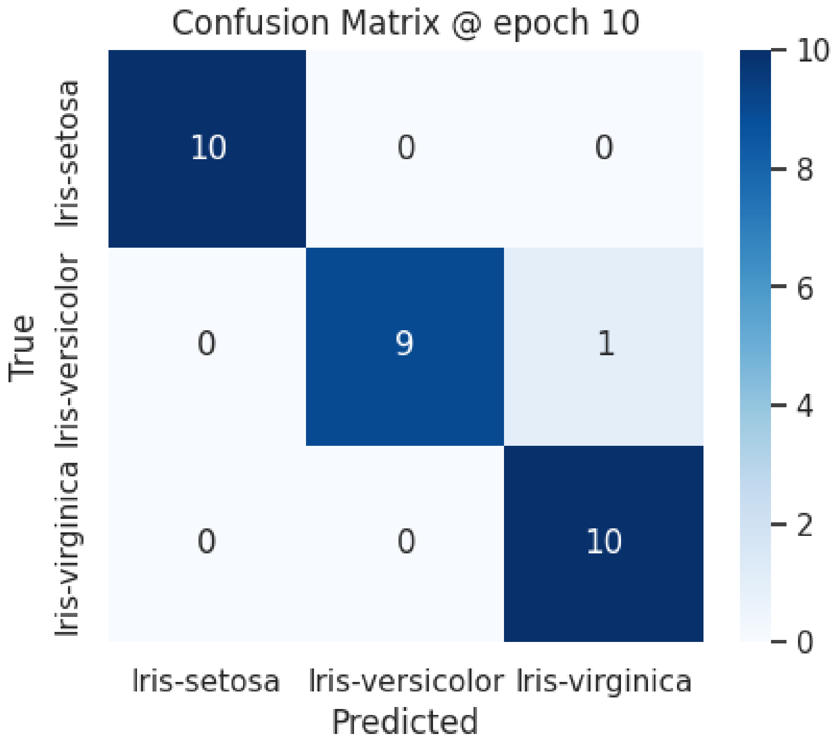
```
Epoch 001 | train_loss=0.7403 train_acc=0.7083 | val_loss=0.5403 val_acc=0.6667
Checkpoint saved: cp_epoch_10.weights.h5 | epoch 10 | train_loss=0.0833
val_acc=0.9667
Epoch 011 | train_loss=0.0718 train_acc=0.9750 | val_loss=0.0604 val_acc=1.0000
Checkpoint saved: cp_epoch_20.weights.h5 | epoch 20 | train_loss=0.0558
val_acc=1.0000
Epoch 021 | train_loss=0.0607 train_acc=0.9667 | val_loss=0.0584 val_acc=0.9667
Checkpoint saved: cp_epoch_30.weights.h5 | epoch 30 | train_loss=0.0500
val_acc=0.9667
Epoch 031 | train_loss=0.0473 train_acc=0.9750 | val_loss=0.0565 val_acc=0.9667
```

```
Checkpoint saved: cp_epoch_40.weights.h5 | epoch 40 | train_loss=0.0441
val_acc=0.9667
Epoch 041 | train_loss=0.0429 train_acc=0.9917 | val_loss=0.0634 val_acc=0.9667
Checkpoint saved: cp_epoch_50.weights.h5 | epoch 50 | train_loss=0.0441
val_acc=0.9667
Epoch 051 | train_loss=0.0574 train_acc=0.9750 | val_loss=0.0808 val_acc=0.9333
Checkpoint saved: cp_epoch_60.weights.h5 | epoch 60 | train_loss=0.0348
val_acc=0.9333
Epoch 061 | train_loss=0.0551 train_acc=0.9750 | val_loss=0.1093 val_acc=0.9667
Checkpoint saved: cp_epoch_70.weights.h5 | epoch 70 | train_loss=0.0371
val_acc=0.9000
Epoch 071 | train_loss=0.0267 train_acc=0.9917 | val_loss=0.0618 val_acc=0.9667
Checkpoint saved: cp_epoch_80.weights.h5 | epoch 80 | train_loss=0.0294
val_acc=0.9667
Epoch 081 | train_loss=0.0399 train_acc=0.9917 | val_loss=0.1152 val_acc=0.9000
Checkpoint saved: cp_epoch_90.weights.h5 | epoch 90 | train_loss=0.0193
val_acc=0.9667
Epoch 091 | train_loss=0.0194 train_acc=0.9917 | val_loss=0.1413 val_acc=0.9000
Checkpoint saved: cp_epoch_100.weights.h5 | epoch 100 | train_loss=0.0299
val_acc=0.9333
Training complete in 64.0s
Saved checkpoints: ['cp_epoch_10.weights.h5', 'cp_epoch_100.weights.h5',
'cp_epoch_20.weights.h5', 'cp_epoch_30.weights.h5', 'cp_epoch_40.weights.h5',
'cp_epoch_50.weights.h5', 'cp_epoch_60.weights.h5', 'cp_epoch_70.weights.h5',
'cp_epoch_80.weights.h5', 'cp_epoch_90.weights.h5']
```

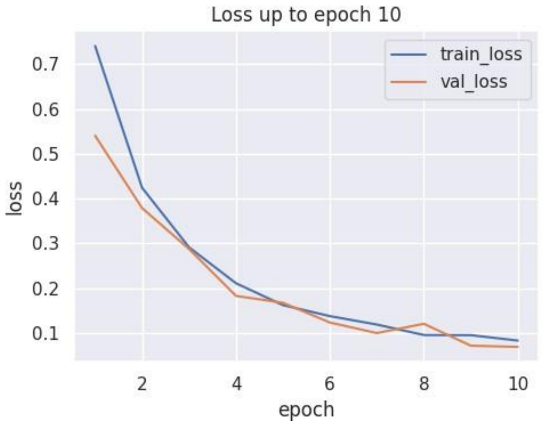
[ ]:

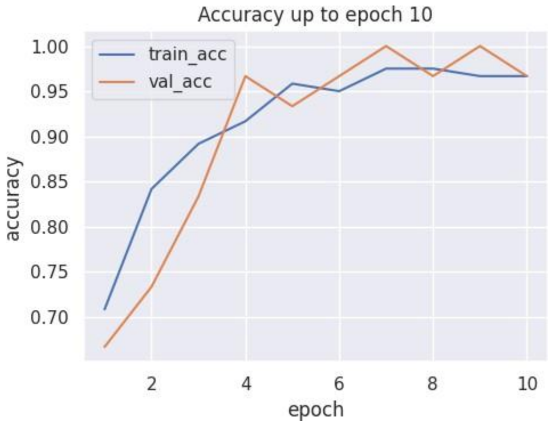
```python
# ---------------- Part 5: Model Evaluation (per-checkpoint outputs) ------------------
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
sns.set()

# Load history and gradnorm files created during training
with open('history_backprop.json') as f:
    history = json.load(f)
with open('gradnorm_history.json') as f:
    grad_hist = json.load(f)

# Sorted checkpoint epochs available
checkpoint_epochs = sorted([int(e) for e in grad_hist.keys()])

def eval_model_at_checkpoint(epoch, model_obj, X_test_arr, y_test_arr, binarizer_obj):
    # Predict and return metrics, confusion matrix & report
    y_probs = model_obj.predict(X_test_arr, verbose=0)
    y_pred = binarizer_obj.inverse_transform(y_probs)
    y_true = binarizer_obj.inverse_transform(y_test_arr)
    acc = accuracy_score(y_true, y_pred)
    cm = confusion_matrix(y_true, y_pred, labels=binarizer_obj.classes_)
    rep = classification_report(y_true, y_pred, output_dict=True)
    return acc, cm, rep, y_true, y_pred

# Evaluate & plot for each checkpoint
for epoch in checkpoint_epochs:
    ckpt_file = f'cp_epoch_{epoch}.weights.h5'
    if not os.path.exists(ckpt_file):
        print("Missing:", ckpt_file); continue

    # Load weights into model
    model.load_weights(ckpt_file)
    print("\n" + "="*70)
    print(f"EVALUATION AT CHECKPOINT: epoch {epoch}")
    print("="*70)

    # 1) Test metrics
    acc, cm, rep_dict, y_true, y_pred = eval_model_at_checkpoint(epoch, model, X_test, y_test, binarizer)
    print(f"Test accuracy @ epoch {epoch}: {acc:.4f}")

    # 2) Confusion matrix
    plt.figure(figsize=(5,4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=binarizer.classes_, yticklabels=binarizer.classes_)
    plt.title(f'Confusion Matrix @ epoch {epoch}')
    plt.xlabel('Predicted');  plt.ylabel('True');  plt.show()

    # 3) Classification report table
    report_df = pd.DataFrame(rep_dict).transpose()
    print("Classification report (precision, recall, f1-score, support):")
    display(report_df)

    # 4) Training curves up to this epoch (PPT style)
    eidx = min(epoch, len(history['train_loss']))
    epochs_range = list(range(1, eidx+1))
    plt.figure(figsize=(10,4))
    plt.subplot(1,2,1)
    plt.plot(epochs_range, history['train_loss'][:eidx], label='train_loss')
    plt.plot(epochs_range, history['val_loss'][:eidx], label='val_loss')
    plt.xlabel('epoch'); plt.ylabel('loss'); plt.title(f'Loss up to epoch {epoch}'); plt.legend()
    plt.subplot(1,2,2)
    plt.plot(epochs_range, history['train_acc'][:eidx], label='train_acc')
    plt.plot(epochs_range, history['val_acc'][:eidx], label='val_acc')
    plt.xlabel('epoch'); plt.ylabel('accuracy'); plt.title(f'Accuracy up to epoch {epoch}'); plt.legend()
    plt.tight_layout(); plt.show()

    # 5) Gradient norms recorded at checkpoint (backprop flow for chosen sample)
    gn = grad_hist[str(epoch)]
    gn_df = pd.DataFrame({'layer': list(gn.keys()), 'grad_L2': list(gn.values())})
    print("Saved gradient L2 norms (per-layer kernel grads) for chosen sample:")
    display(gn_df)

    plt.figure(figsize=(6,3))
    plt.bar(gn_df['layer'], gn_df['grad_L2'])
    plt.title(f'Gradient kernel L2 norms @ epoch {epoch} (chosen sample)')
    plt.ylabel('L2 norm'); plt.show()

print("\nAll checkpoint evaluations complete.")
```

```
======================================================================
EVALUATION AT CHECKPOINT: epoch 10
======================================================================
Test accuracy @ epoch 10: 0.9667
```

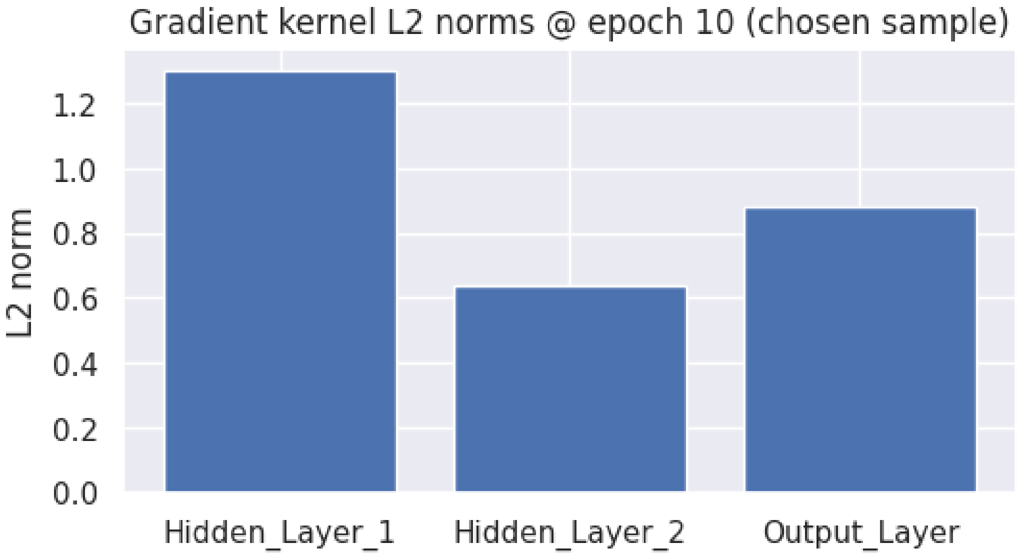


```
Classification report (precision, recall, f1-score, support):
                  precision    recall  f1-score    support
Iris-setosa        1.000000  1.000000  1.000000  10.000000
Iris-versicolor    1.000000  0.900000  0.947368  10.000000
Iris-virginica     0.909091  1.000000  0.952381  10.000000
accuracy           0.966667  0.966667  0.966667   0.966667
macro avg          0.969697  0.966667  0.966583  30.000000
weighted avg       0.969697  0.966667  0.966583  30.000000
```

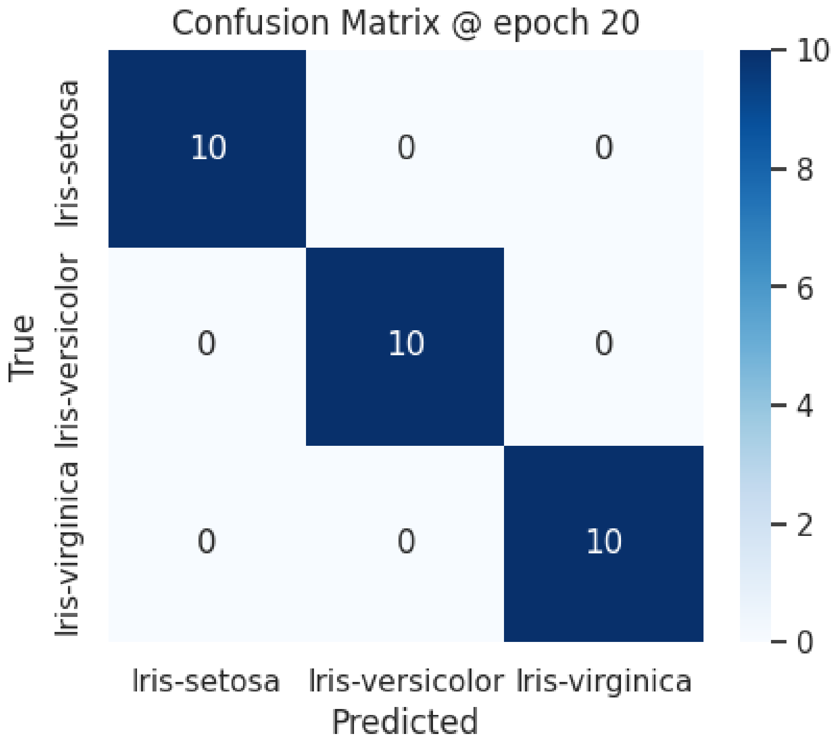




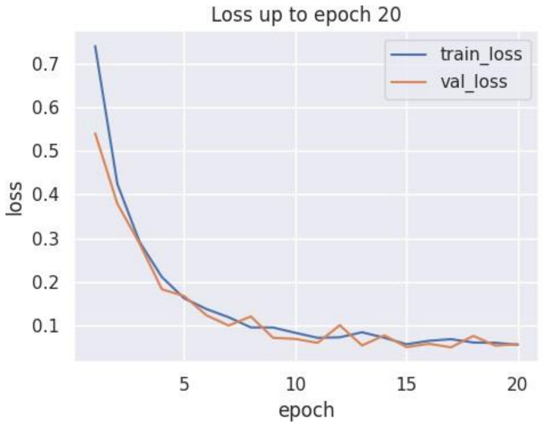
```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
           layer   grad_L2
0  Hidden_Layer_1  1.303545
1  Hidden_Layer_2  0.640402
2    Output_Layer  0.882976
```

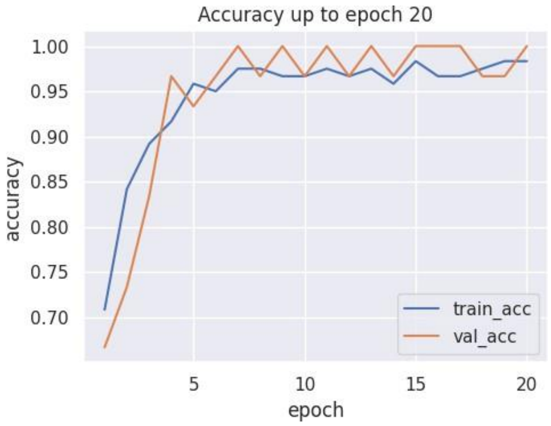


```
======================================================================
EVALUATION AT CHECKPOINT: epoch 20
======================================================================
Test accuracy @ epoch 20: 1.0000
```

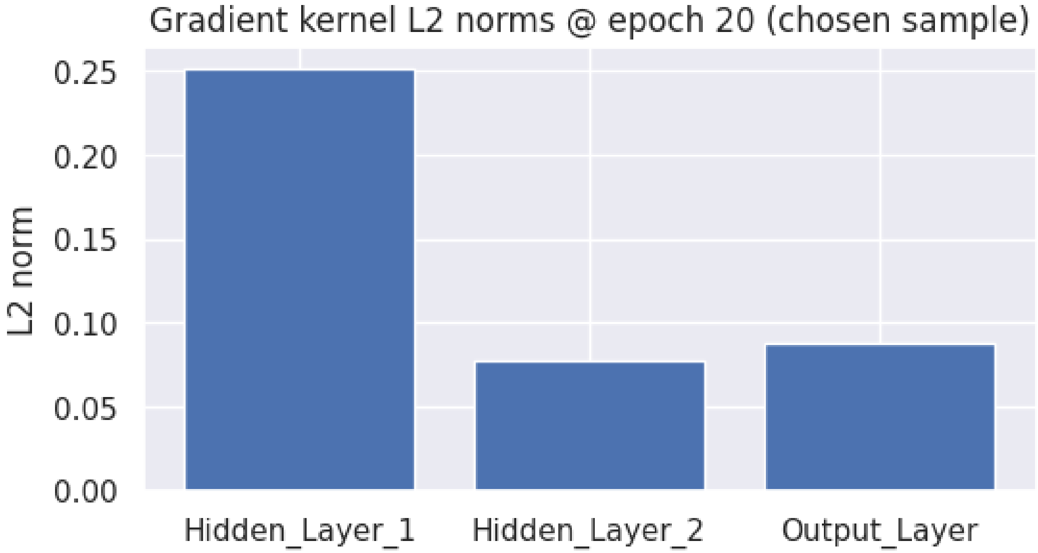


```
Classification report (precision, recall, f1-score, support):

                 precision   recall    f1-score  support
Iris-setosa            1.0      1.0         1.0     10.0
Iris-versicolor        1.0      1.0         1.0     10.0
Iris-virginica         1.0      1.0         1.0     10.0
accuracy               1.0      1.0         1.0      1.0
macro avg              1.0      1.0         1.0     30.0
weighted avg           1.0      1.0         1.0     30.0
```

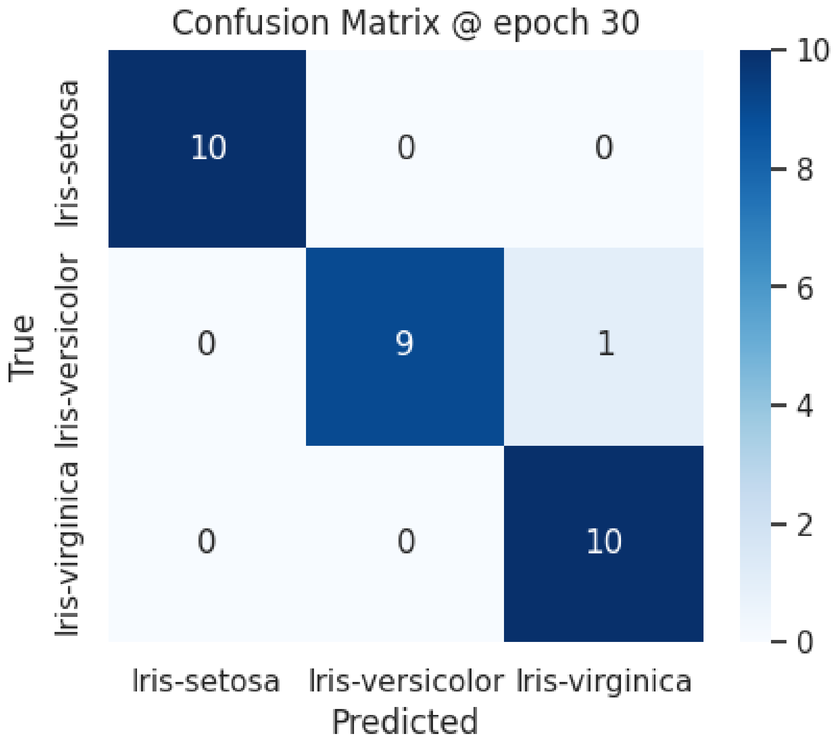




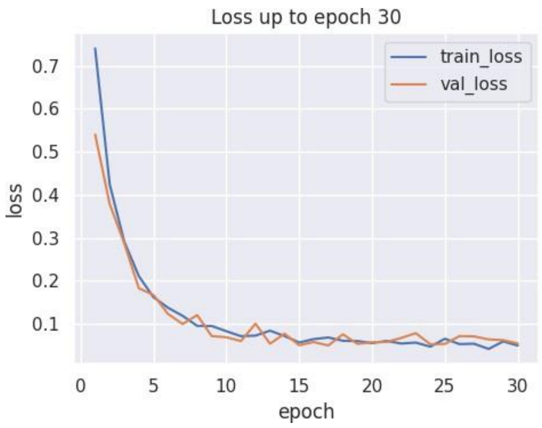
```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
           layer   grad_L2
0  Hidden_Layer_1  0.251714
1  Hidden_Layer_2  0.077331
2    Output_Layer  0.087632
```

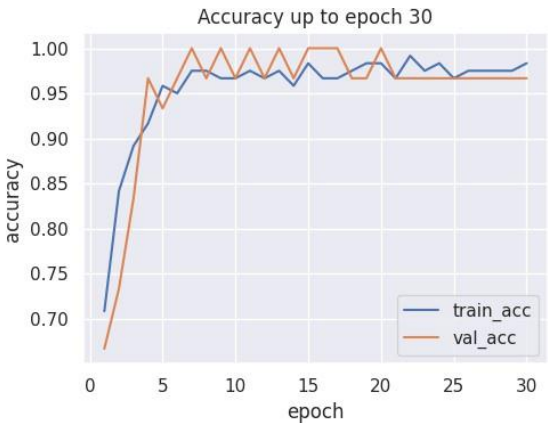


```
============================================================================
EVALUATION AT CHECKPOINT: epoch 30
============================================================================
Test accuracy @ epoch 30: 0.9667
```

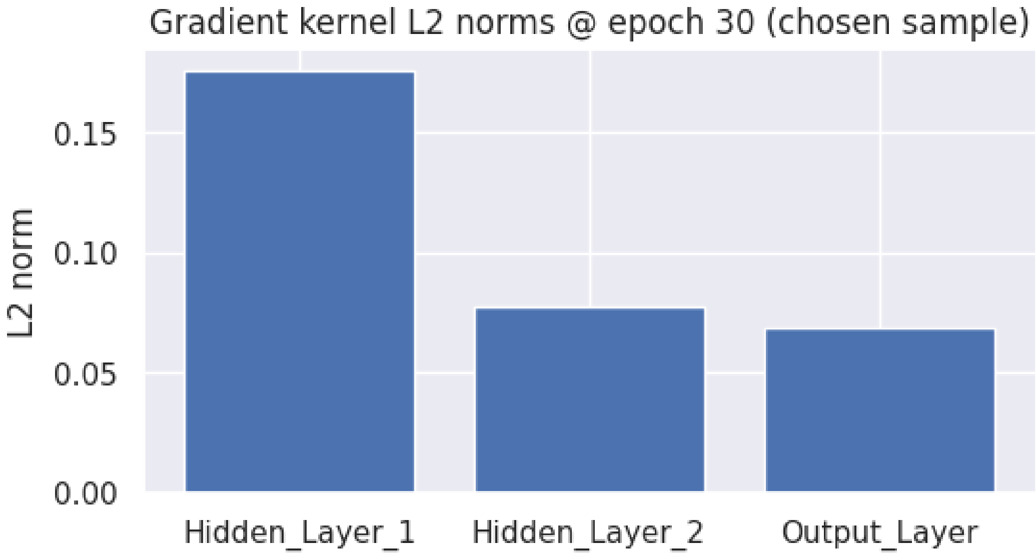


```
Classification report (precision, recall, f1-score, support):
                 precision    recall  f1-score    support
Iris-setosa       1.000000  1.000000  1.000000  10.000000
Iris-versicolor   1.000000  0.900000  0.947368  10.000000
Iris-virginica    0.909091  1.000000  0.952381  10.000000
accuracy          0.966667  0.966667  0.966667   0.966667
macro avg         0.969697  0.966667  0.966583  30.000000
weighted avg      0.969697  0.966667  0.966583  30.000000
```

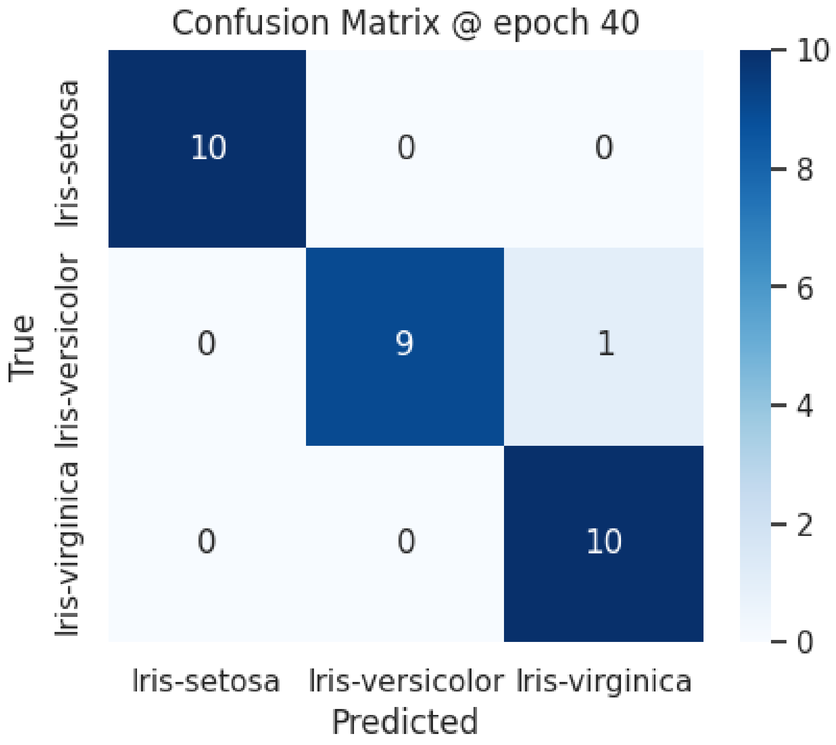




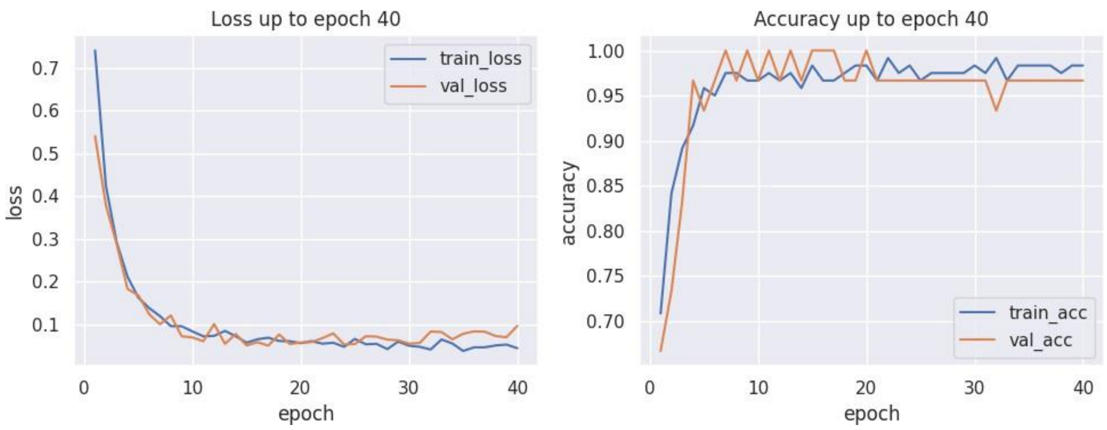
```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
            layer   grad_L2
0  Hidden_Layer_1  0.176141
1  Hidden_Layer_2  0.076963
2    Output_Layer  0.068207
```



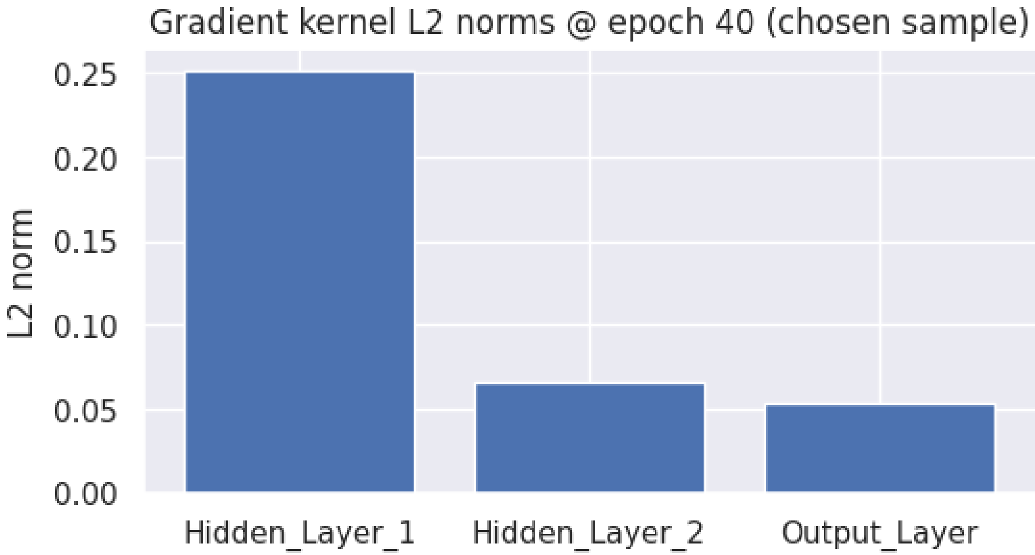
```
======================================================================
EVALUATION AT CHECKPOINT: epoch 40
======================================================================
Test accuracy @ epoch 40: 0.9667
```



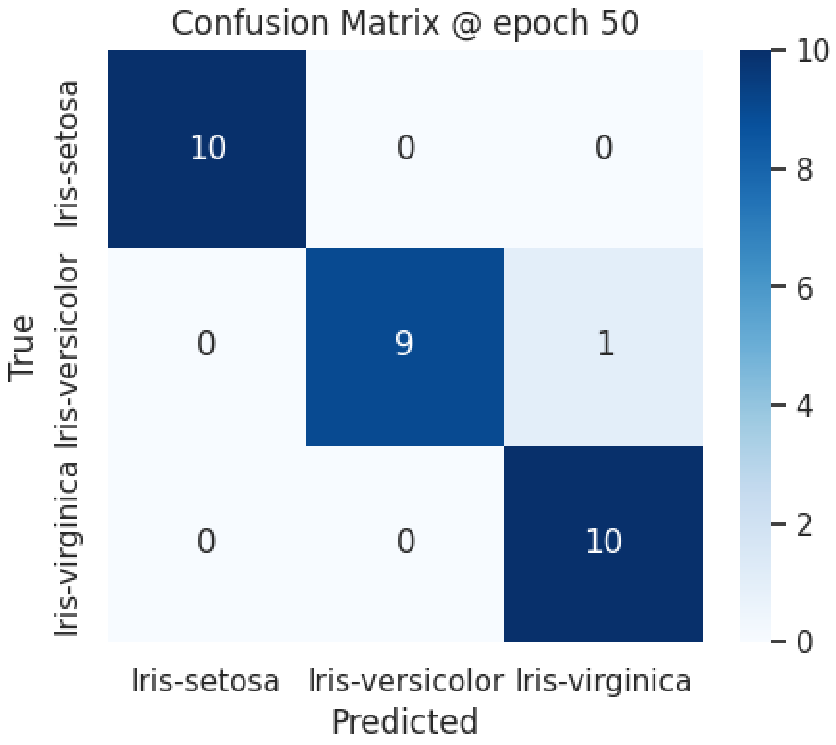
```
Classification report (precision, recall, f1-score, support):
                precision    recall  f1-score    support
Iris-setosa      1.000000  1.000000  1.000000  10.000000
Iris-versicolor  1.000000  0.900000  0.947368  10.000000
Iris-virginica   0.909091  1.000000  0.952381  10.000000
accuracy         0.966667  0.966667  0.966667   0.966667
macro avg        0.969697  0.966667  0.966583  30.000000
weighted avg     0.969697  0.966667  0.966583  30.000000
```



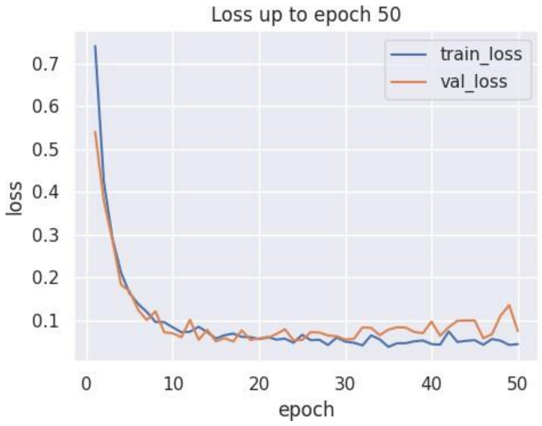
```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
           layer   grad_L2
0  Hidden_Layer_1  0.251897
1  Hidden_Layer_2  0.066437
2    Output_Layer  0.053002
```

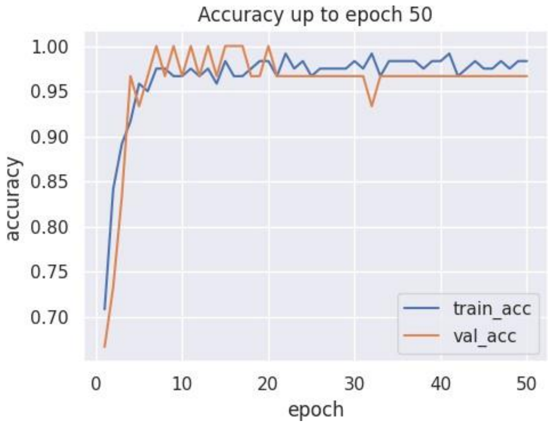


```
======================================================================
EVALUATION AT CHECKPOINT: epoch 50
======================================================================
Test accuracy @ epoch 50: 0.9667
```

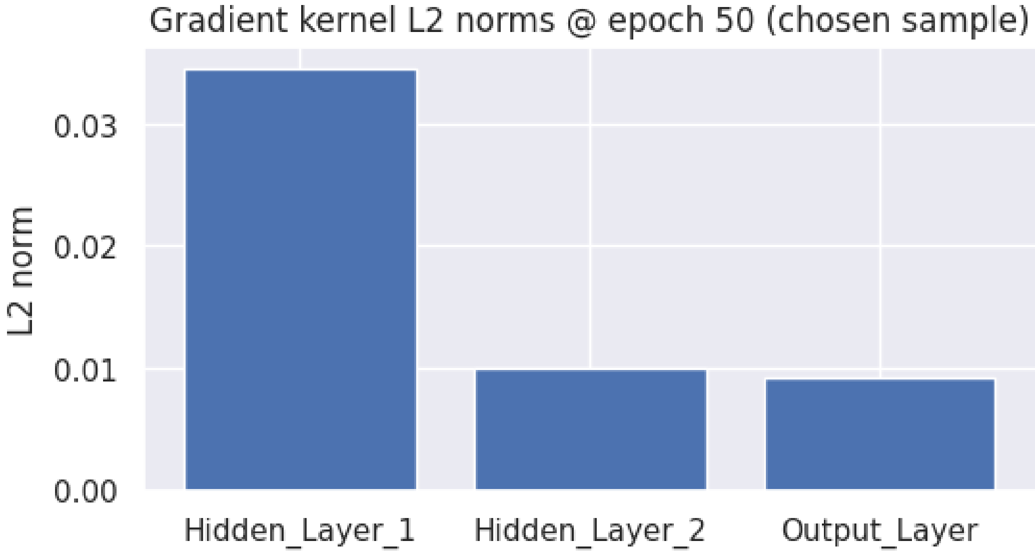


```
Classification report (precision, recall, f1-score, support):
                  precision    recall  f1-score    support
Iris-setosa        1.000000  1.000000  1.000000  10.000000
Iris-versicolor    1.000000  0.900000  0.947368  10.000000
Iris-virginica     0.909091  1.000000  0.952381  10.000000
accuracy           0.966667  0.966667  0.966667   0.966667
macro avg          0.969697  0.966667  0.966583  30.000000
weighted avg       0.969697  0.966667  0.966583  30.000000
```

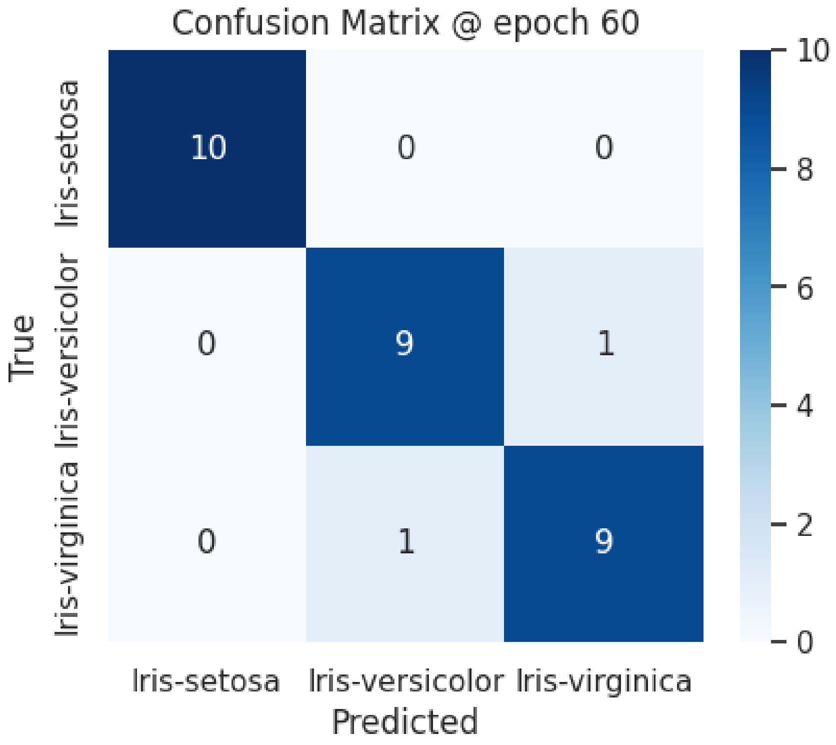




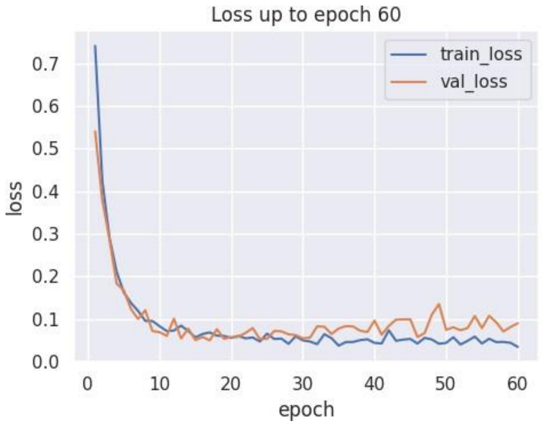
```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
            layer   grad_L2
0  Hidden_Layer_1  0.034616
1  Hidden_Layer_2  0.010067
2    Output_Layer  0.009264
```

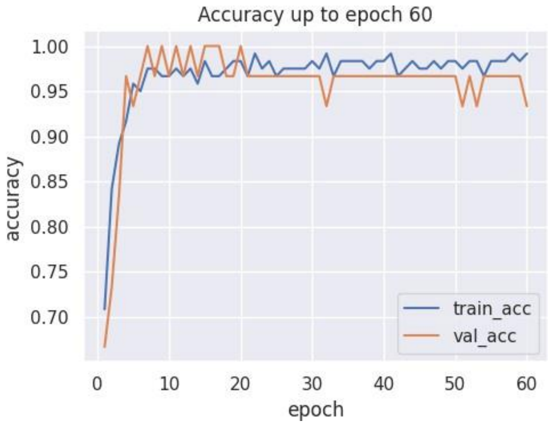


```
======================================================================
EVALUATION AT CHECKPOINT: epoch 60
======================================================================
Test accuracy @ epoch 60: 0.9333
```

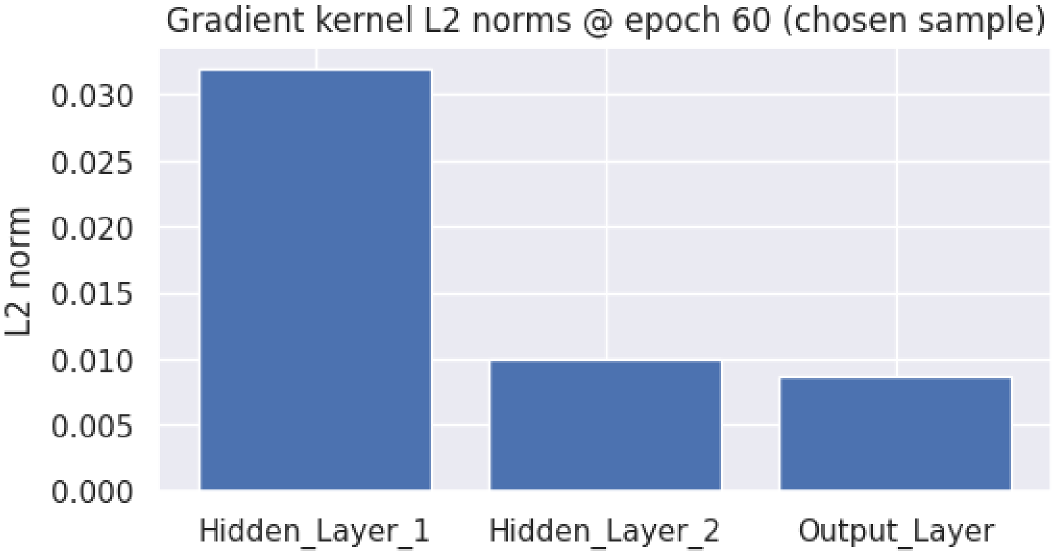


```
Classification report (precision, recall, f1-score, support):
                 precision    recall  f1-score    support
Iris-setosa       1.000000  1.000000  1.000000  10.000000
Iris-versicolor   0.900000  0.900000  0.900000  10.000000
Iris-virginica    0.900000  0.900000  0.900000  10.000000
accuracy          0.933333  0.933333  0.933333   0.933333
macro avg         0.933333  0.933333  0.933333  30.000000
weighted avg      0.933333  0.933333  0.933333  30.000000
```

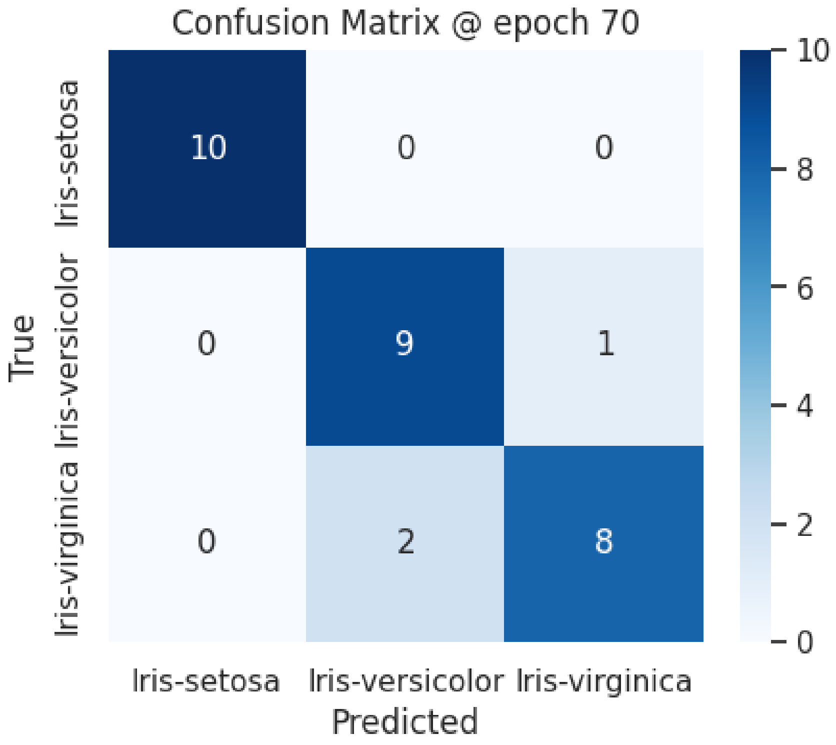




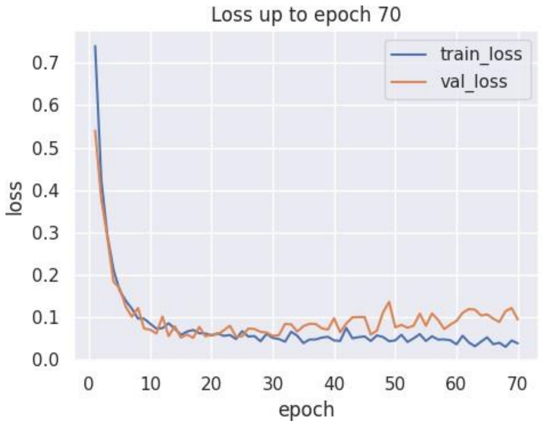
```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
            layer   grad_L2
0  Hidden_Layer_1  0.032021
1  Hidden_Layer_2  0.009950
2    Output_Layer  0.008729
```

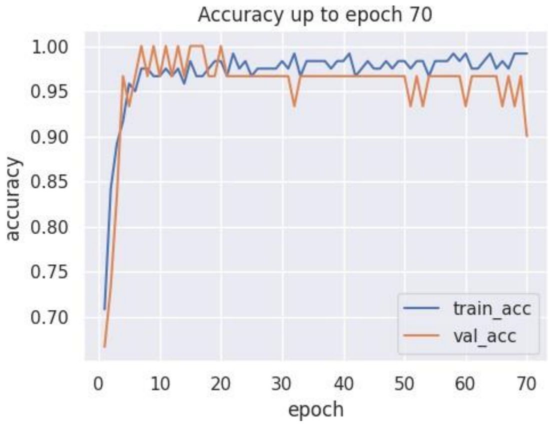


```
======================================================================
EVALUATION AT CHECKPOINT: epoch 70
======================================================================
Test accuracy @ epoch 70: 0.9000
```

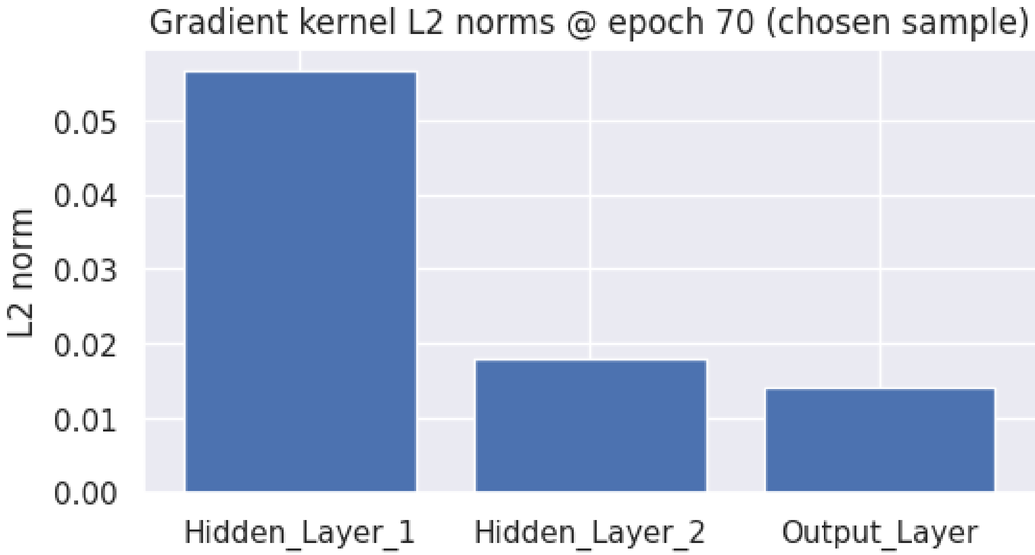


```
Classification report (precision, recall, f1-score, support):
                  precision  recall  f1-score  support
Iris-setosa        1.000000     1.0  1.000000     10.0
Iris-versicolor    0.818182     0.9  0.857143     10.0
Iris-virginica     0.888889     0.8  0.842105     10.0
accuracy           0.900000     0.9  0.900000      0.9
macro avg          0.902357     0.9  0.899749     30.0
weighted avg       0.902357     0.9  0.899749     30.0
```

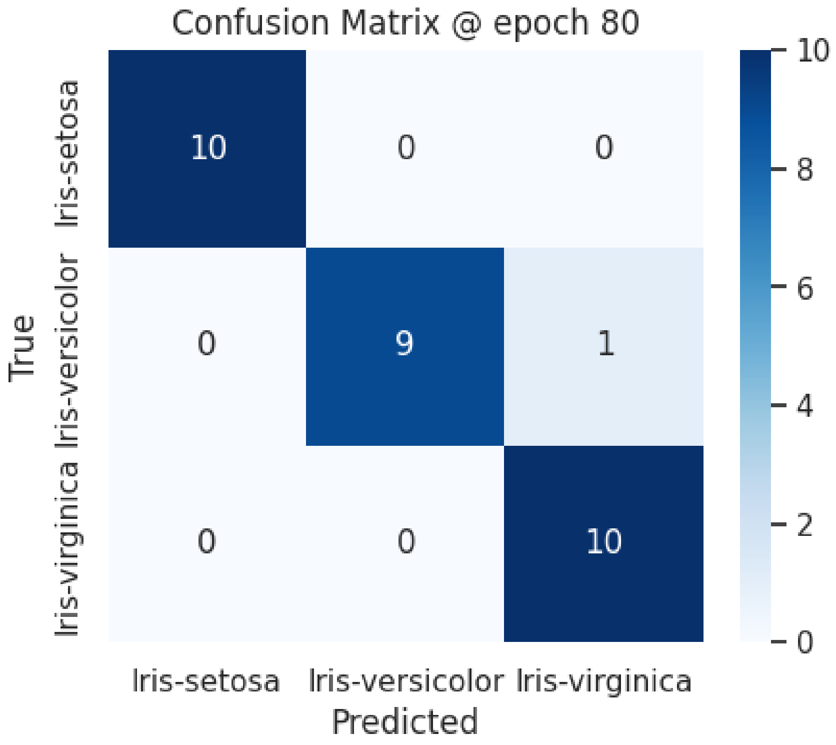


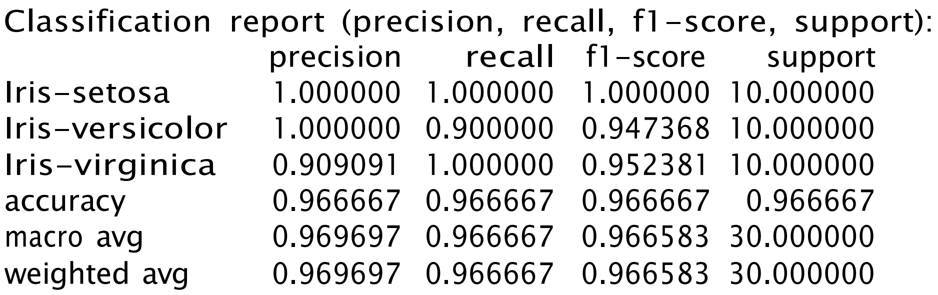


```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
            layer   grad_L2
0  Hidden_Layer_1  0.056764
1  Hidden_Layer_2  0.018078
2    Output_Layer  0.014204
```

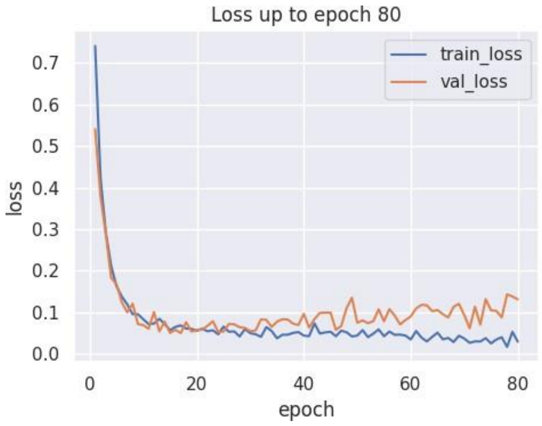


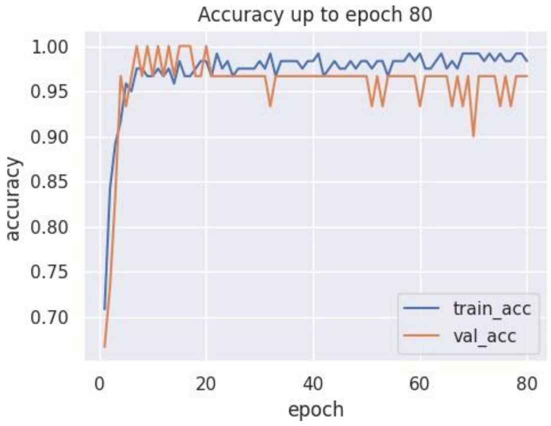
```
======================================================================
EVALUATION AT CHECKPOINT: epoch 80
======================================================================
Test accuracy @ epoch 80: 0.9667
```



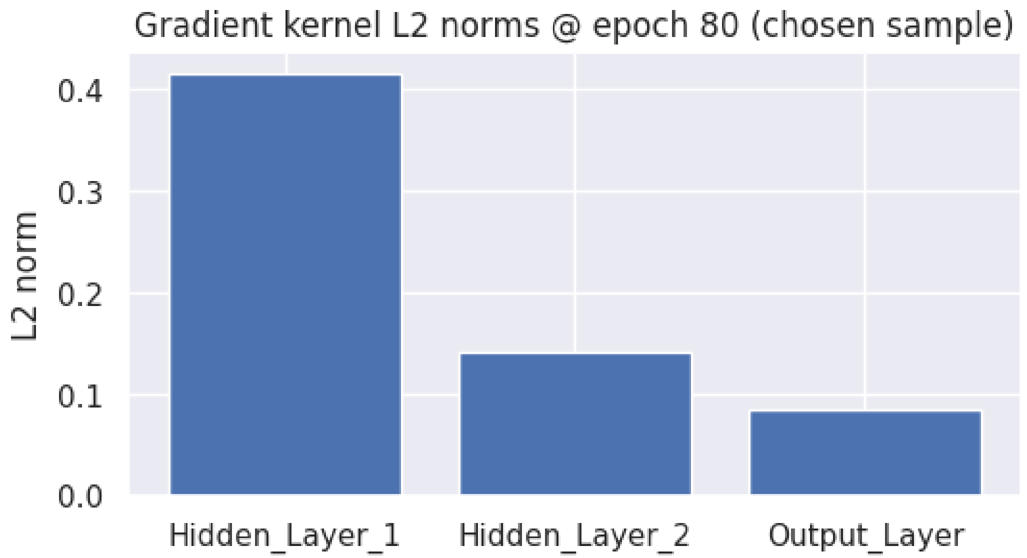


```
Classification report (precision, recall, f1−score, support):
                    precision    recall  f1−score    support
Iris−setosa          1.000000  1.000000  1.000000  10.000000
Iris−versicolor      1.000000  0.900000  0.947368  10.000000
Iris−virginica       0.909091  1.000000  0.952381  10.000000
accuracy             0.966667  0.966667  0.966667   0.966667
macro avg            0.969697  0.966667  0.966583  30.000000
weighted avg         0.969697  0.966667  0.966583  30.000000
```

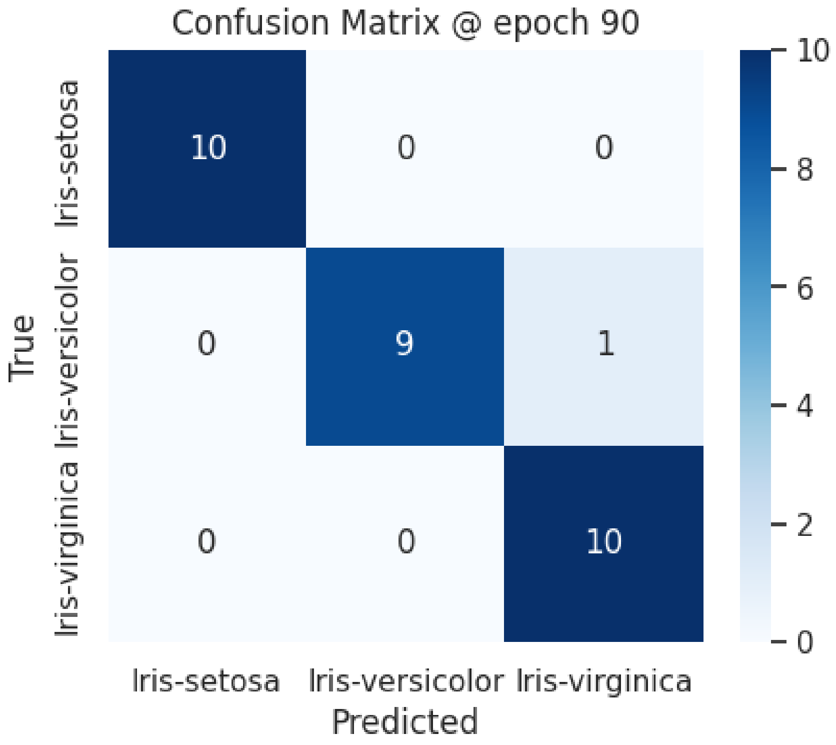




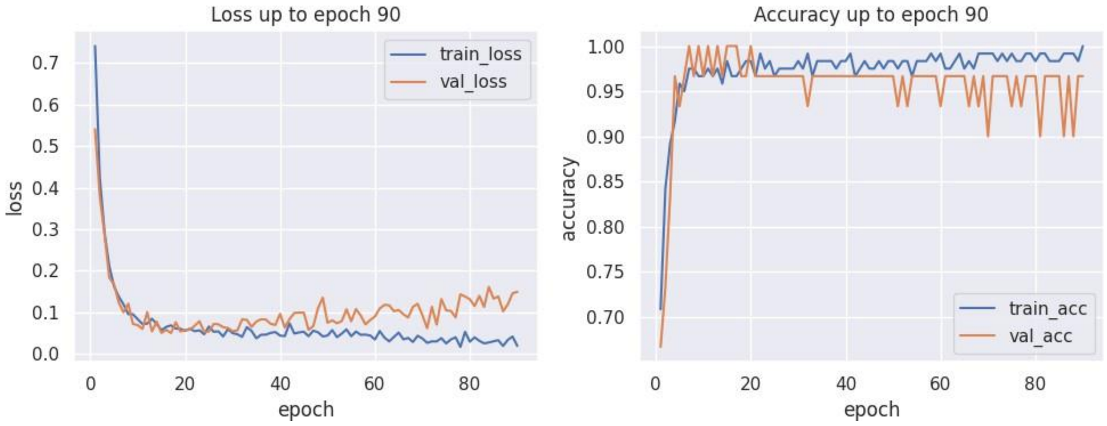
```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
            layer   grad_L2
0  Hidden_Layer_1  0.416494
1  Hidden_Layer_2  0.141671
2    Output_Layer  0.084102
```



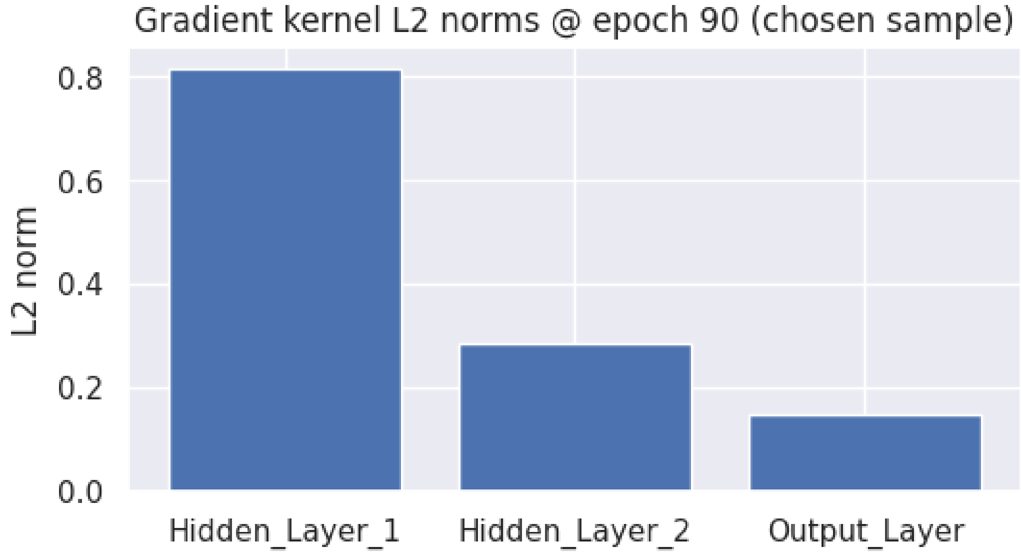
```
======================================================================
EVALUATION AT CHECKPOINT: epoch 90
======================================================================
Test accuracy @ epoch 90: 0.9667
```



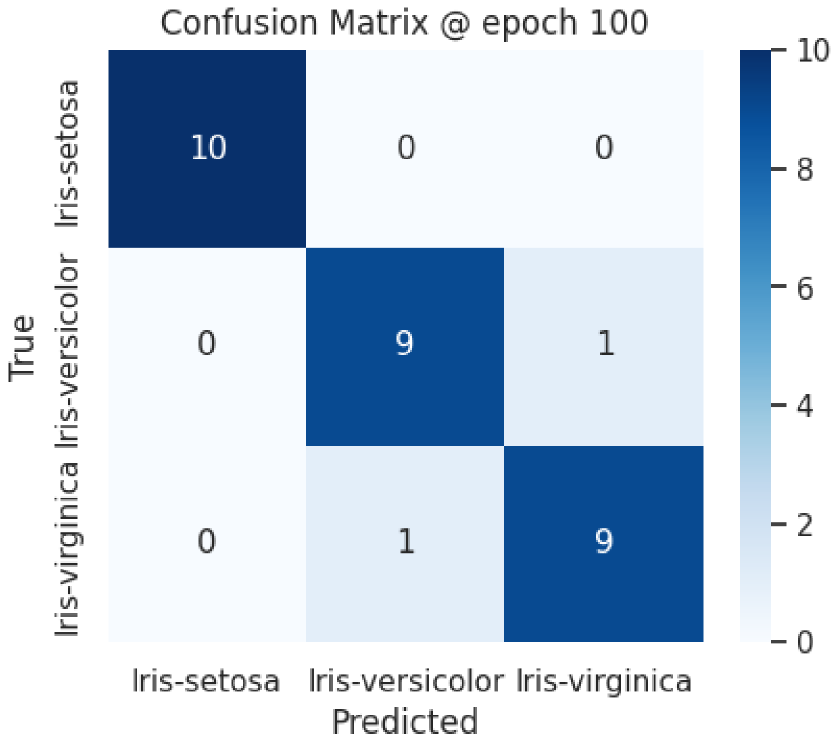
```
Classification report (precision, recall, f1-score, support):
                  precision    recall  f1-score     support
Iris-setosa        1.000000  1.000000  1.000000   10.000000
Iris-versicolor    1.000000  0.900000  0.947368   10.000000
Iris-virginica     0.909091  1.000000  0.952381   10.000000
accuracy           0.966667  0.966667  0.966667    0.966667
macro avg          0.969697  0.966667  0.966583   30.000000
weighted avg       0.969697  0.966667  0.966583   30.000000
```



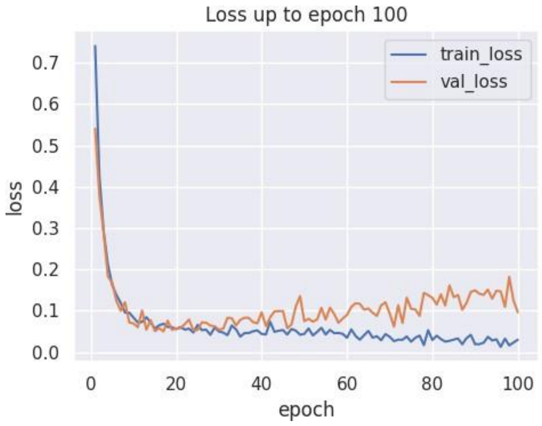
```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
            layer   grad_L2
0  Hidden_Layer_1  0.815791
1  Hidden_Layer_2  0.282643
2    Output_Layer  0.148078
```

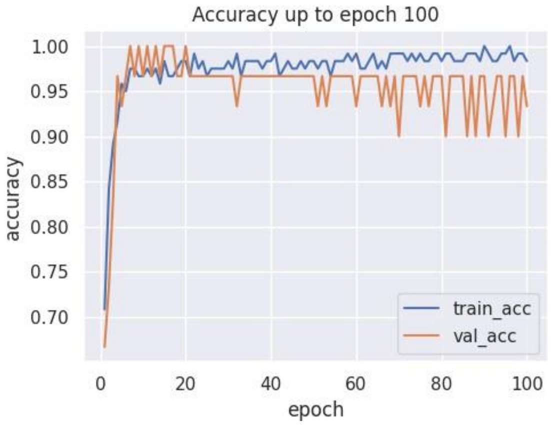


```
======================================================================
EVALUATION AT CHECKPOINT: epoch 100
======================================================================
Test accuracy @ epoch 100: 0.9333
```

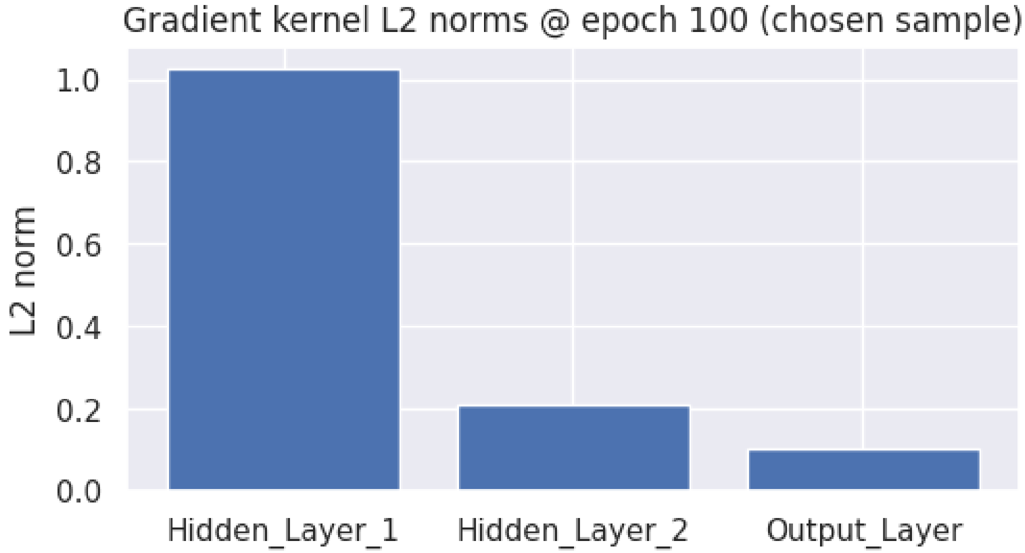


```
Classification report (precision, recall, f1-score, support):
                  precision    recall  f1-score    support
Iris-setosa        1.000000  1.000000  1.000000  10.000000
Iris-versicolor    0.900000  0.900000  0.900000  10.000000
Iris-virginica     0.900000  0.900000  0.900000  10.000000
accuracy           0.933333  0.933333  0.933333   0.933333
macro avg          0.933333  0.933333  0.933333  30.000000
weighted avg       0.933333  0.933333  0.933333  30.000000
```





```
Saved gradient L2 norms (per-layer kernel grads) for chosen sample:
           layer   grad_L2
0  Hidden_Layer_1  1.028418
1  Hidden_Layer_2  0.208460
2    Output_Layer  0.103101
```



```
All checkpoint evaluations complete.
```